

Figure 1

Modified SSA-conversion process

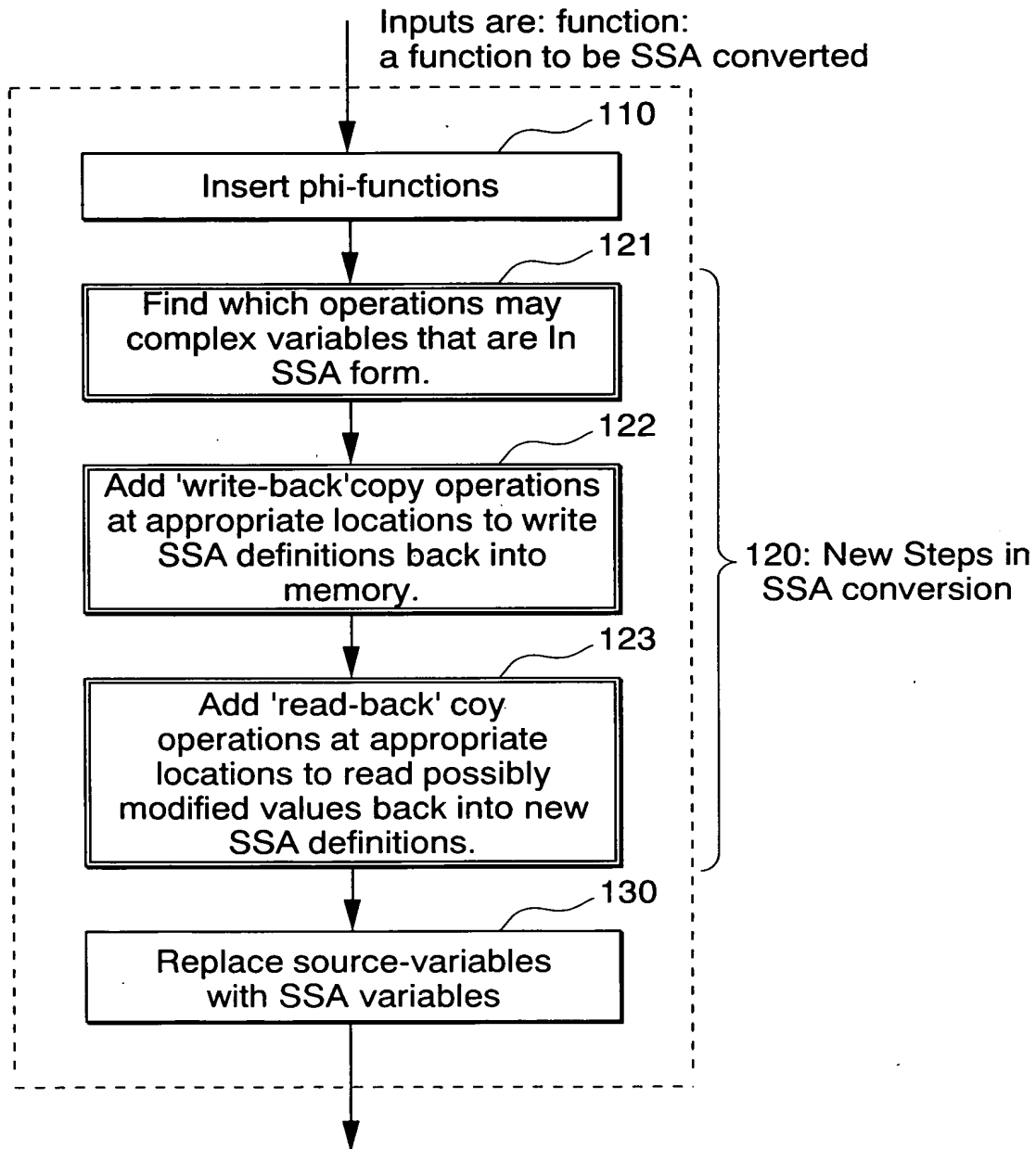


Figure 2
Overall compiler control flow

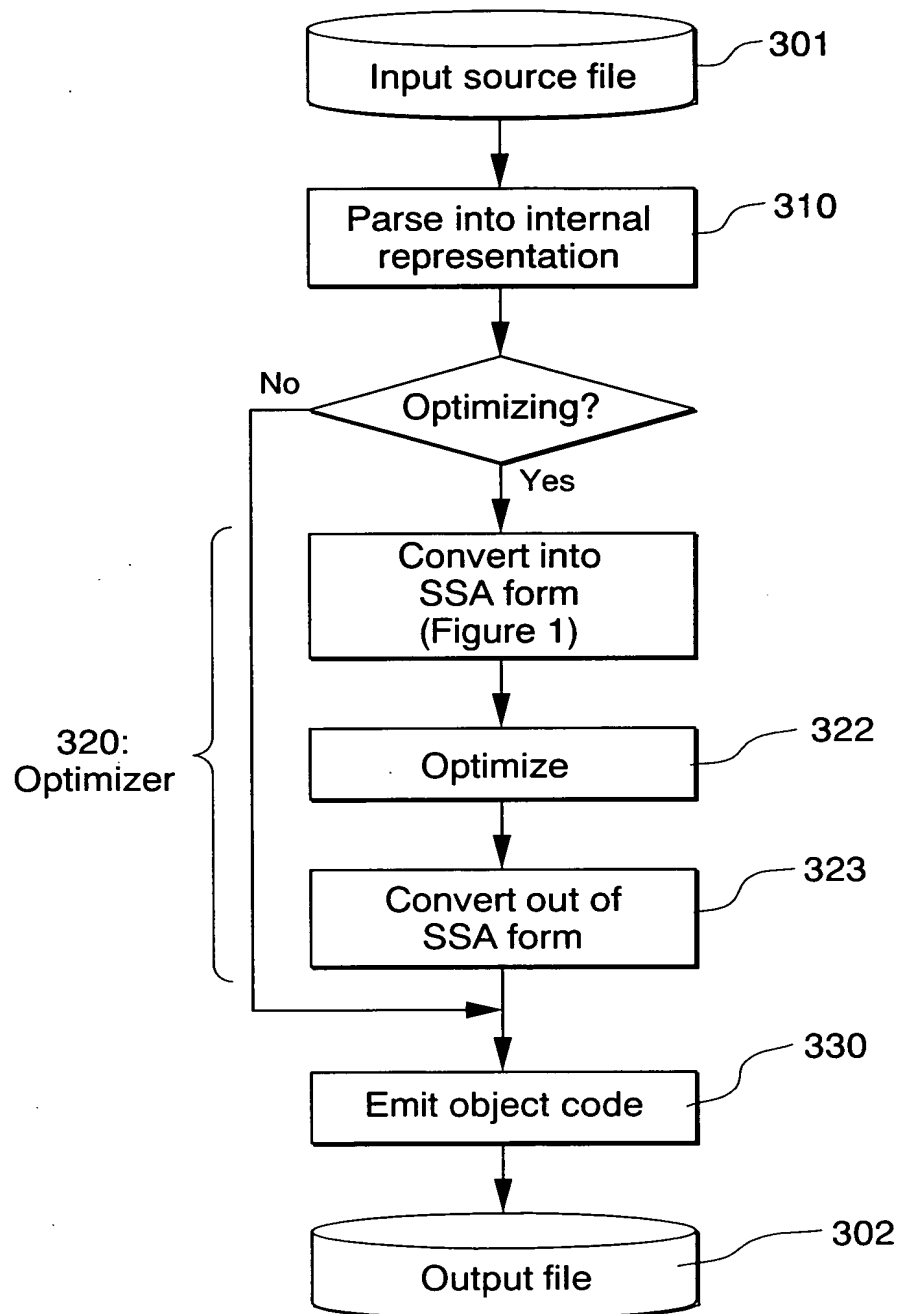


Figure 3

Program representation

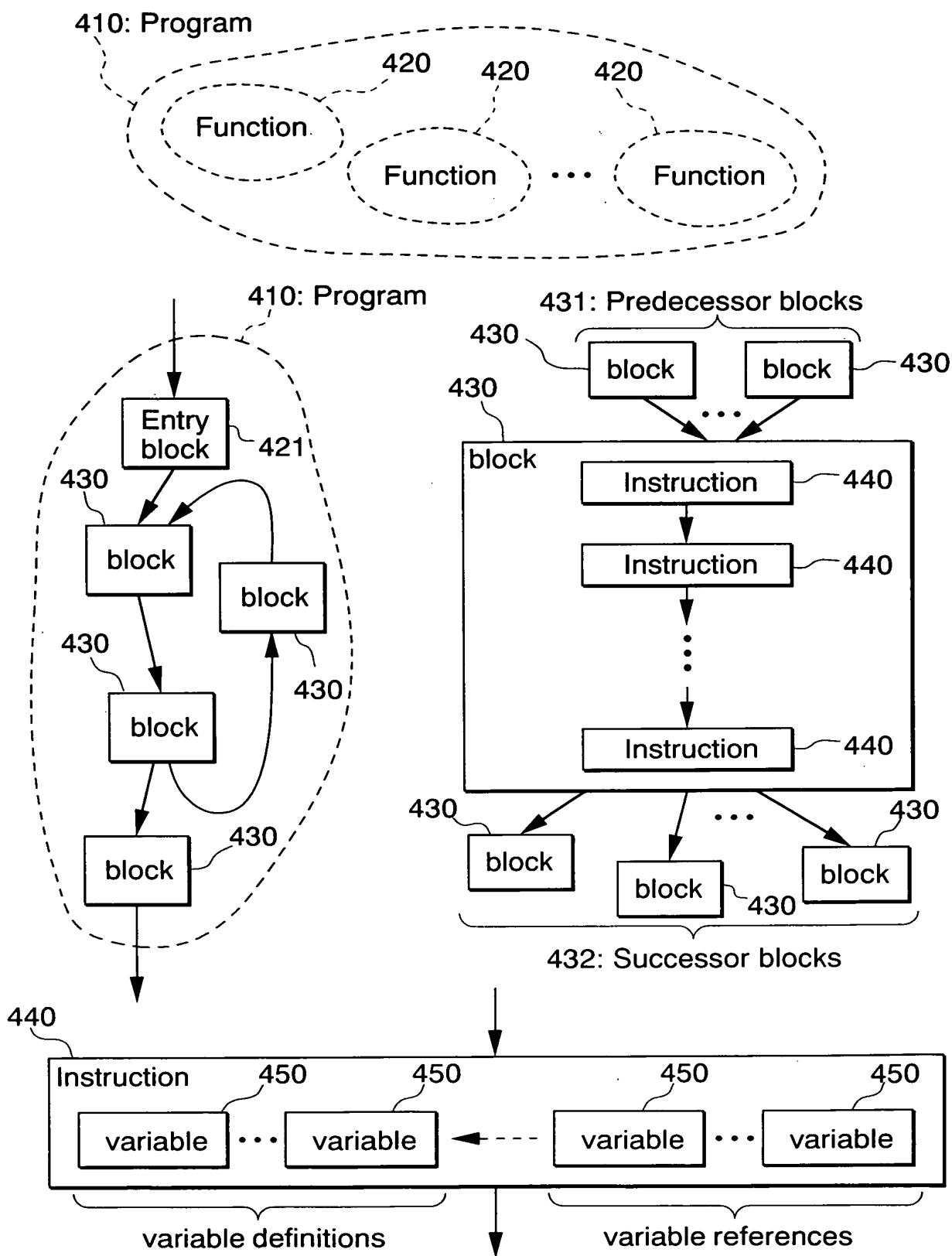
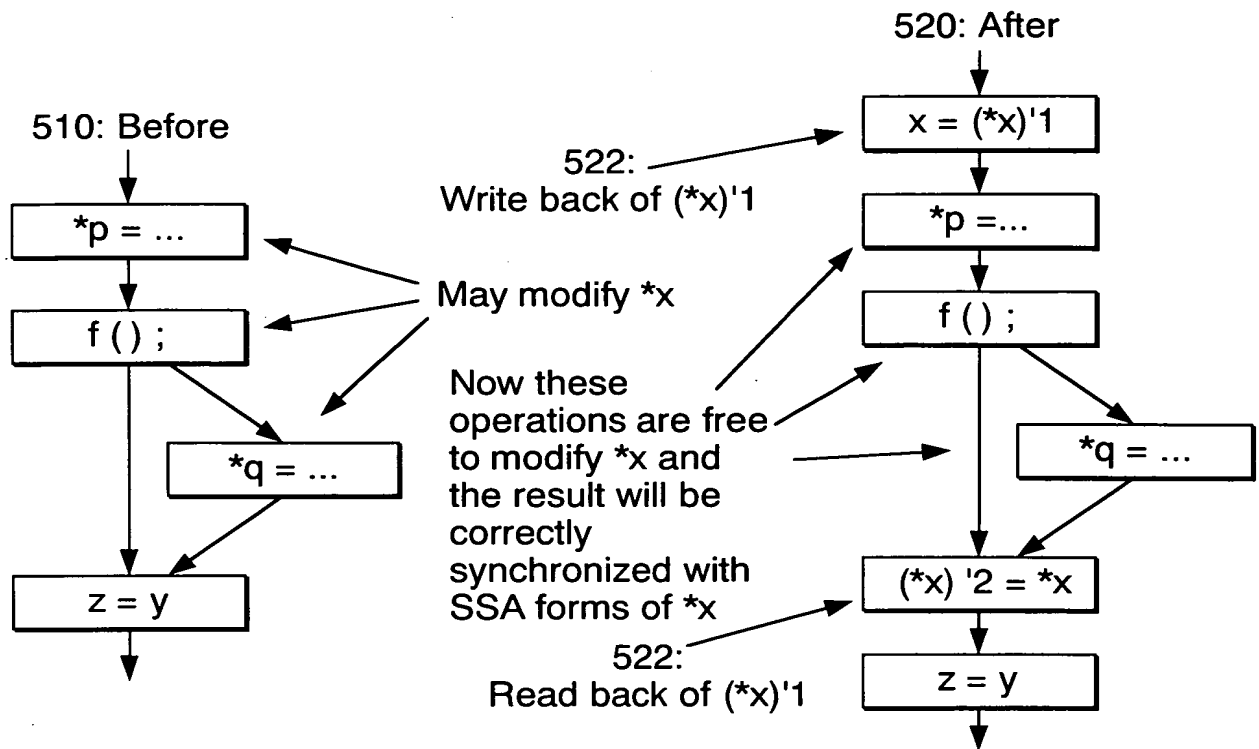


Figure 4

Placement of read/write-backs for the SSA form of $*x$, $(*x)'1$



530: A more naive method for synchronization introduces many read/write-backs

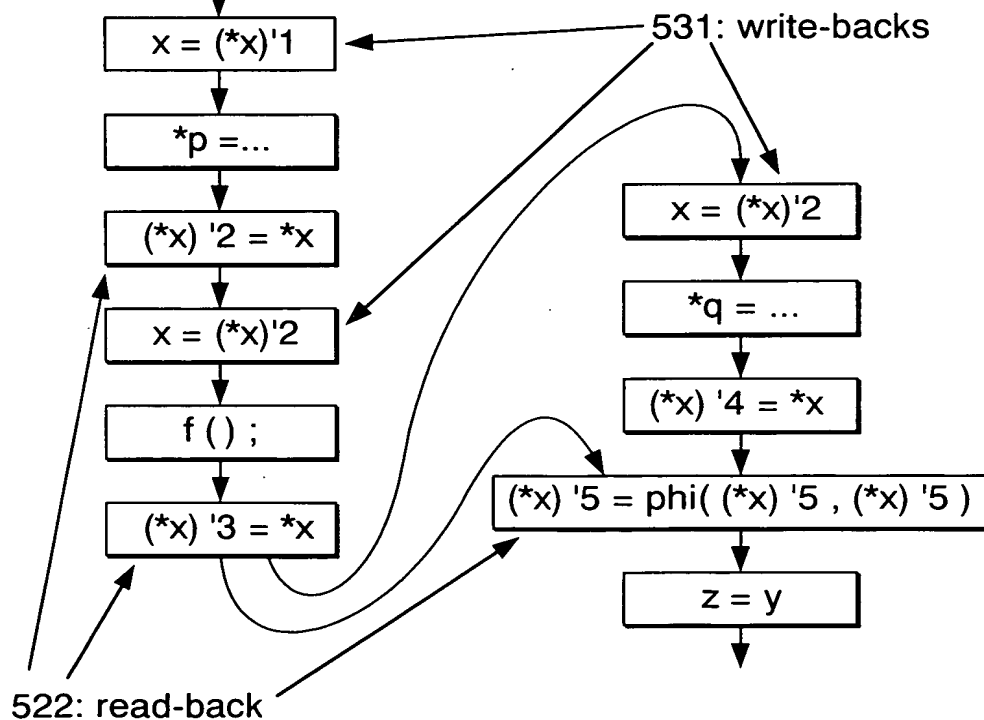


Figure 5

The procedure 'add_syncs_and_write_backs'

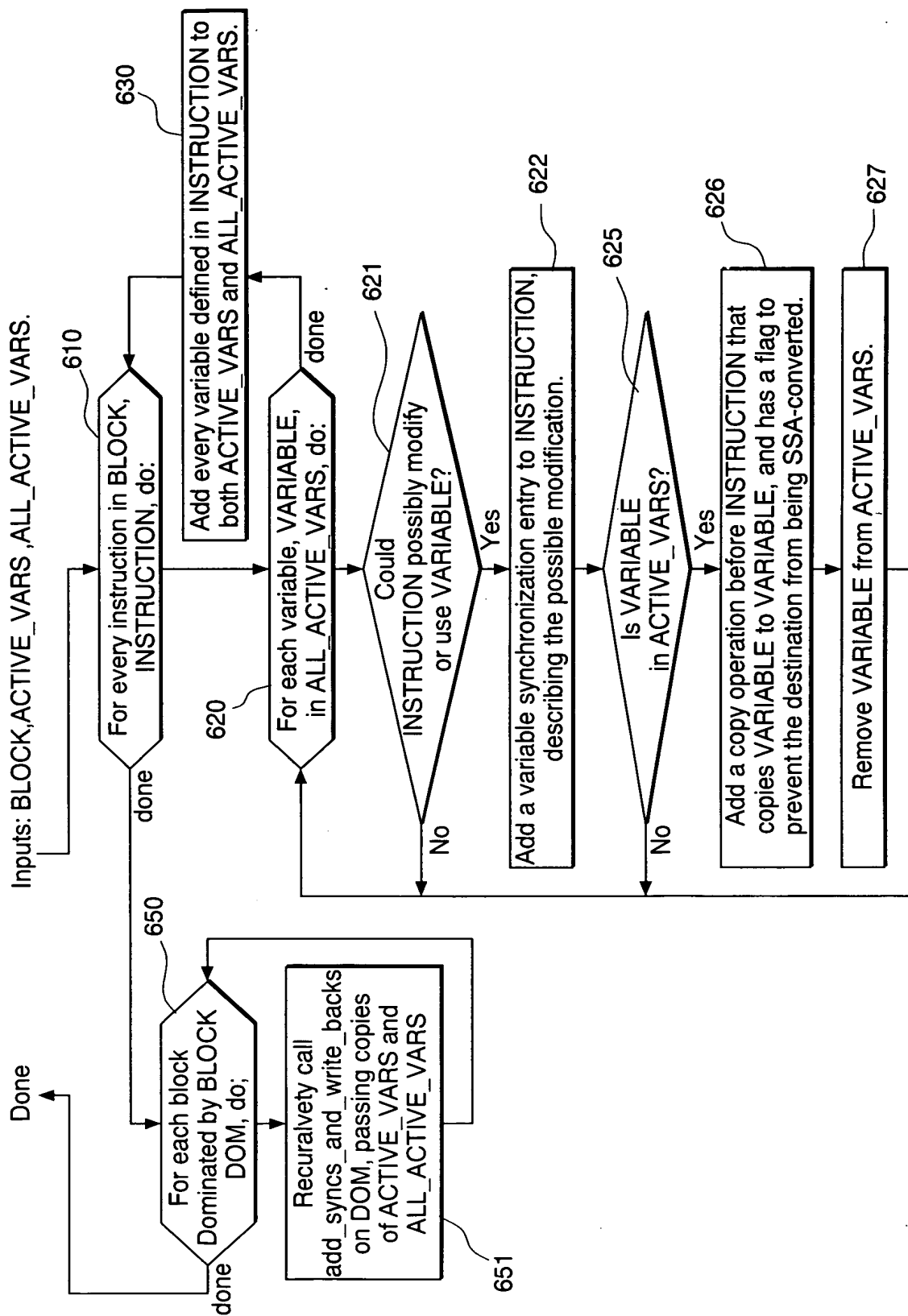


Figure 6

Conversion step (a'.III), insertion of read-backs

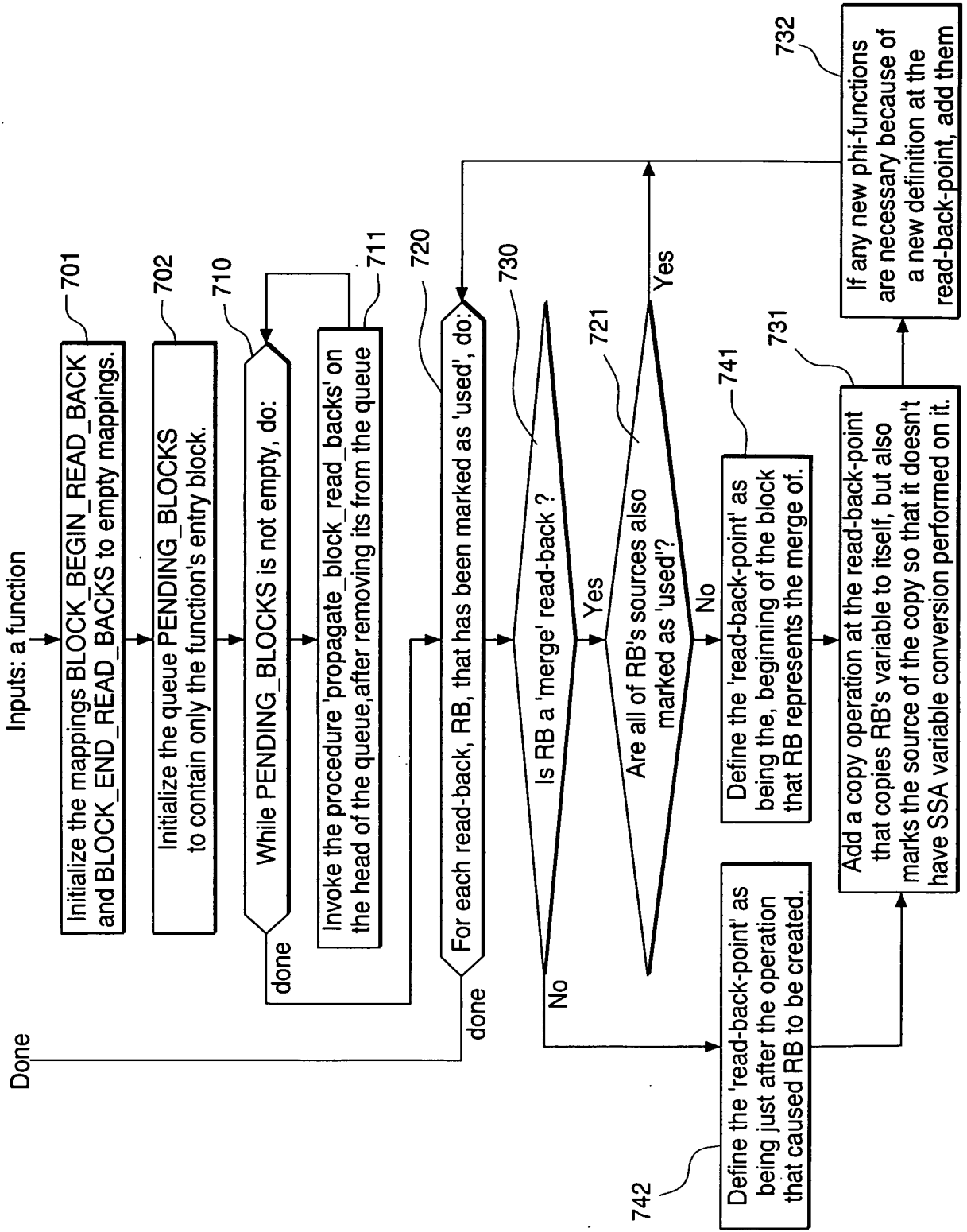


Figure 7A
The procedure 'propagate_read_backs'

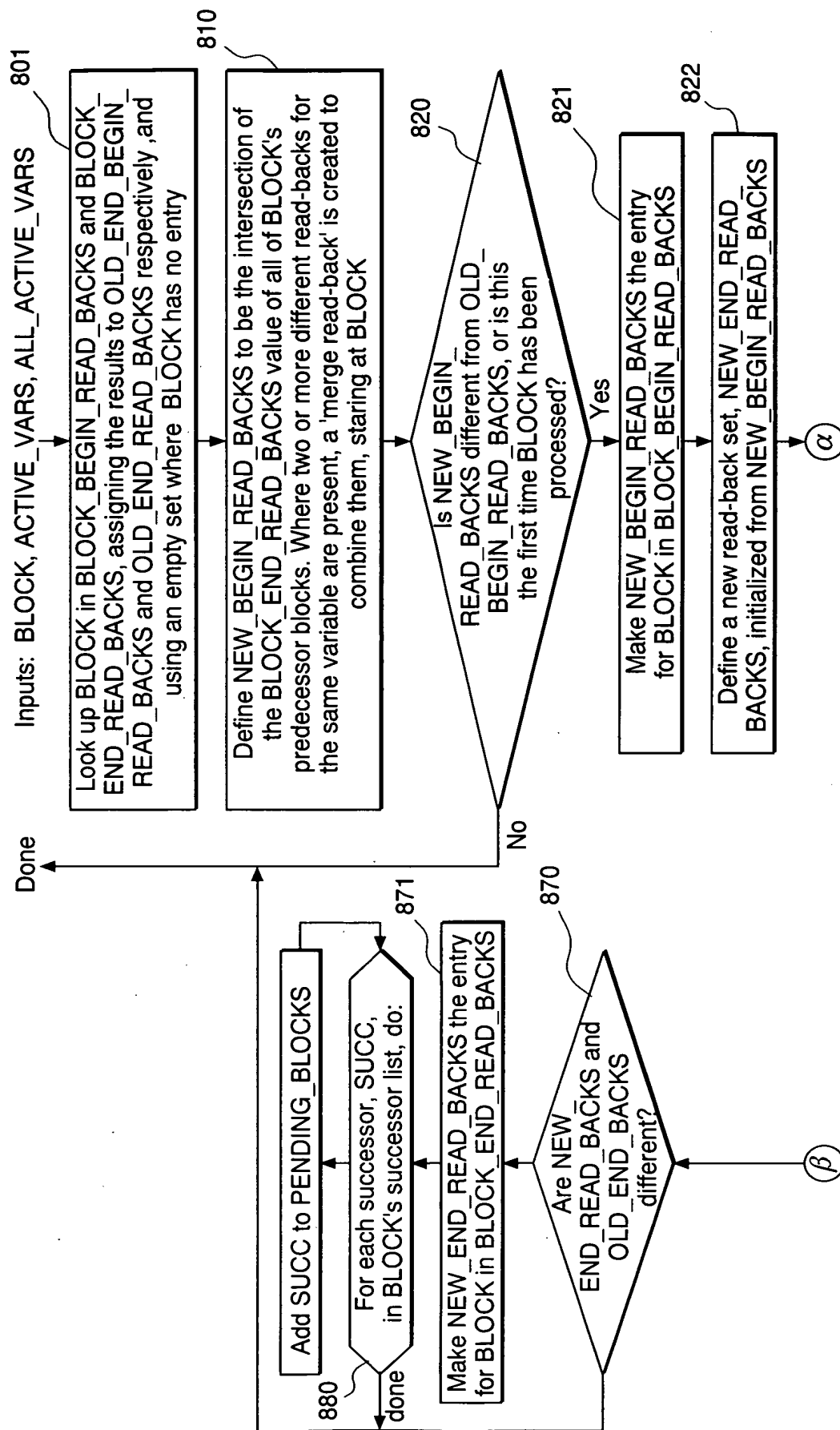
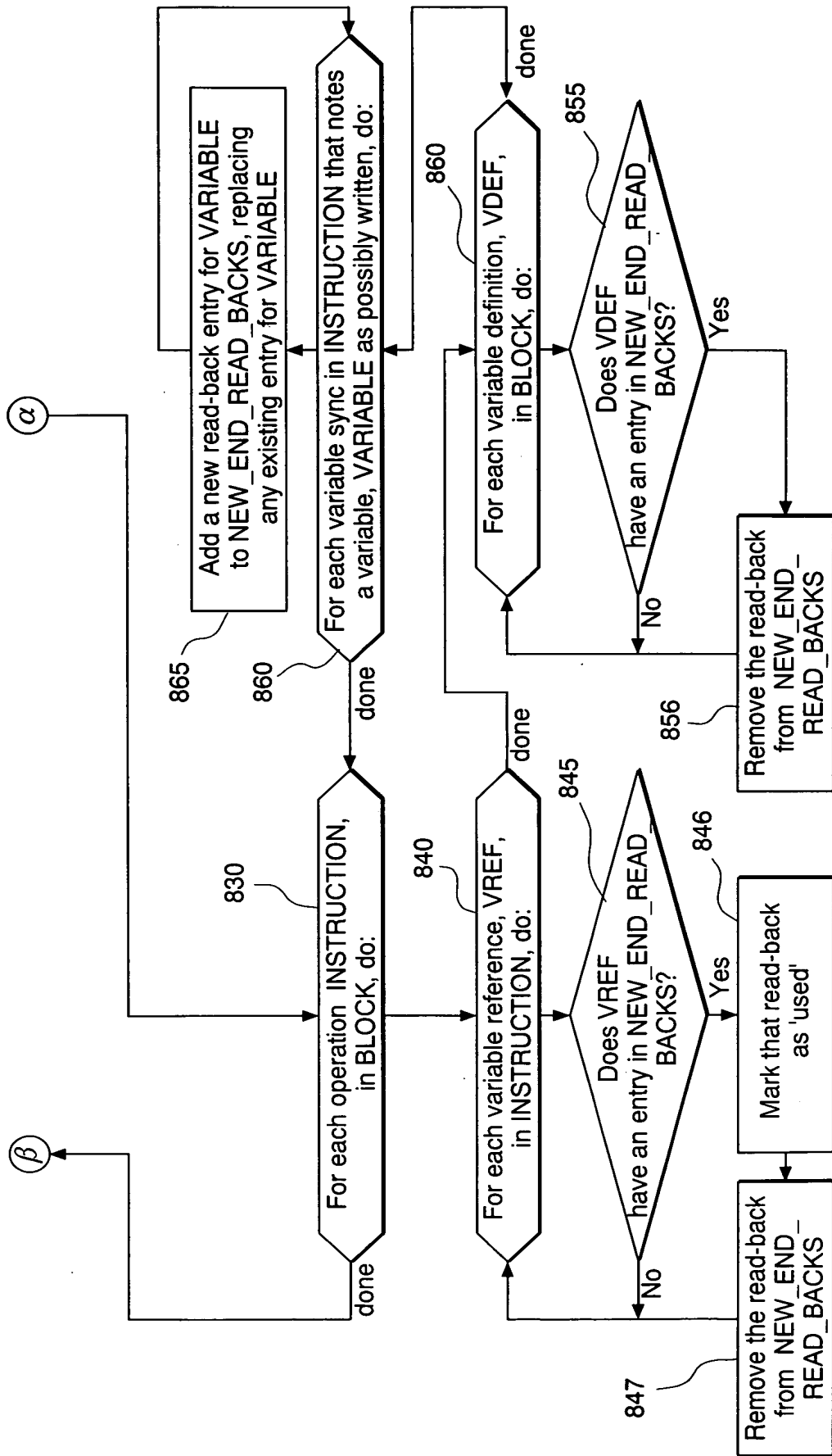


Figure 7B
The procedure 'propagate_read_backs'



SSA converted program, with simple implementation of read-backs :

The following is psuedo-C, augmented with the ϕ operation, where

RESULT = phi (block1: VAL1 , ..., blockN:VALN)

means 'assign VAL1 to RESULT if control-flow comes from block1 ', and similarly so on for each value of N.

The extra variables 'pvN', where N is an integer, are SSA versions of *P, and are in fact local variables, not dereferences of p.

```

1 int foo (int *p)
{
    int pv1, pv2 , pv3 , pv4, pv5, pv6;

    block1;
    pv1 = *P + 1;
    if (pv1 <= 10)
        goto block8;
    block2:
    *P = pv1;          /* This writes-back PV1 to *P. */
    g ();
    pv2 = *P;          /* This reads-back *P into PV2. */
    *P = pv2;          /* This writes-back PV2 to *P. */
    h ( ) ;
    pv3 = *P;          /* This reads-back *P into PV3 */
    if (x <= 5)
        goto block4;
    block3:
    *p = pv3;          /* This writes-back PV4 to *p, */
    g ( ) ;
    pv4 = *p;          /* This reads-back *p into PV4. */
    block4:
    pv5 = phi (block3: pv4, block2: pv3)
    if (x > 3)
        goto block6;
    block5:
    goto block7;
    block6;
    i ( ) ;
    block7:
    x = phi (block6: x, block5: pv5);
    block8:
    pv6 = phi (block1: pv1, block7: 5);
    *P = pv6;          /* This writes-back PV6 to *P. */
    return pv6;
}

```

Figure 10

SSA converted program,
with the implementation of read-backs described in this patent

```
int foo (int *p)
{
    int pv1, pv2, pv3;

    block1:
        pv1 = *p +1;
        if (pv1 <= 10)
            goto block8;

    block2;
        *p = pv1;          /* This writes-back pv1 to *P. */
        g ();
        h ();
        if (x <= 5)
            goto block4;

    block3;
        g ();

    block4:
        pv2 = *p;          /* This reads-back *p into pv2, */
        if (x > 3)
            goto block6;

    block5;
        goto block7;

    block6:
        i ();

    block7 :
        x = phi (block6 : x, block5 : pv2) ;

    block8:
        pv3 = phi (block1: pv1, block7: 5);
        *P = pv3;          /*This writes-back PV3 to *P */

    return pv3;
}
```

[1011]
[1021]
[1022]
[1023]
[1030]
[1024]
[1031]
[1010]

Figure 11

Register-allocated and SSA-unconverted program

using BBA-form requires having a good register allocator that will merge variables where possible, as it tends to generate a lot of variables with short lifetimes. We assume that here.

```
int foo (int *p)
{
    int pv;

    block1;
    pv = *p + 1;
    if (pv <= 10)
        goto block8;

    block2:
    'P = pv;          /* This writes-back pv to *P. */
    g ();
    h ();
    if (x <= 5)
        goto block4;

    block3 :
    g ();

    block4 :
    if (x > 3)
        goto block6;

    block5:
    x=*p;
    goto block7;

    block6:
    i ();

    block7:
    pv =5;

    block8:
    *P= pv          /* This writes-back PV to *P. */

    return pv;
}
```

Figure 12

Original SSA-conversion process

